Assimilator Documentation

Release 1.0

Nicolas Videla

Sep 27, 2017

Contents

JSON REST API wrapper for vendor firewalls.

CHAPTER 1

Introduction

Assimilator was built to enable automotion through REST API, using JSON objects for easy understanding. With Assimilator a developer can self-serve his/her access through the network, while also auditors can request information without the need of network engineers. This API wraps around all possible vendor Firewalls, let it be appliances, virtual machines or cloud ACL.

With Assimilator one can automatize Firewall rules easily, just by simply make an HTTP request one can add/remove/modify/view rules and routes.

MIT License

Altough there are other repos and people working on Firewall automation, this is the only repo that serves an API. I'm currently working alone in this and that's why I released Assimilator under MIT License, because I need other people to help with other Firewall brands and bug fixes.

Assimilator License

CHAPTER 2

Install

Assimilator can be installed through Docker or cloned into a directory and run from there. Personally I prefer Docker since it's more reliable, but both ways work.

The Docker Way

The best way to install Assimilator is through Docker:

```
$ docker pull videlanicolas/assimilator:stable
```

The latest build is constantly improving, I recomend the stable version or instead the latest tag which are also stable:

```
$ docker pull videlanicolas/assimilator:1.2.2
```

Run a container:

```
$ docker run -d -v /path/to/configuration:/etc/assimilator/ -p 443:443 videlanicolas/assimilator:stable
```

Docker containers are not peristent, so if you want to maintain your configured Firewalls and API keys you should mount an external directory into the container, that's what the -v is for.

The Repo-Cloning Way

A.K.A I don't trust your Docker image.

You can clone the repo from Github and build your image of Assimilator from the dockerfile.

```
$ git clone https://github.com/videlanicolas/assimilator.git $ docker build -t assimilator .
```

If you don't want to use Docker there is a bash script to install the dependencies. Also there is another bash script to generate a random certificate for HTTPS connections.

```
$ git clone https://github.com/videlanicolas/assimilator.git $ chmod +x install.sh generate_certificate.sh $
./generate_certificate.sh $ ./install.sh
```

CHAPTER 3

First steps

The first thing you need to do is create a configuration file that adjusts to your needs. Many of these parameters have already been configured for you, but some minimal configuration is needed.

An example configuration file can be found in the repo. This file specifies the initial configuration for Assimilator, this should be mounted as a volume in the Docker container with the '-v' argument on '/etc/assimilator/'.

General

Logfile indicates where logs should be stored.

    logfile = /var/log/assimilator.log

The log level that should be logged [DEBUG, INFO, WARN, ERROR, CRIT, FATAL].

    loglevel = WARN

The date and time format for the logs, the default is Syslog friendly.

    format = %d/%m/%Y %H:%M:%S

The location for the API keys of each user, this file should exist only. API keys are managed through the REST api.

    apikeyfile = /etc/assimilator/api.key

The location for all Firewall related authentication. This is managed thorugh the REST api.

    firewalls = /etc/assimilator/firewalls.json

Where the API should listen.

    address = 0.0.0.0

What port should Assimilator listen to, default is 443.

    port = 443

Key Management

This is the authentication required to modify Firewall credentials and user's API keys.

From where should Assimilator authenticate users? For now, the only option is 'static'.

type = static

The user and password required for admin login to the API.

user = admin password = secret

Firewall Management

Same as Key Management, this section describes the admin user and password required to configure Firewall credentials.

From where should Assimilator authenticate users? For now, the only option is 'static'.

type = static

The user and password required for admin login to the API.

user = admin password = secret

CHAPTER 4

API Key Management

There are two URL from where the admin logs in, one of those is the Key management.

Key management handles the API keys sent to Assimilator, it identifies API keys with a matching authorization token. When an API key is randomly generated it has no authorization to do stuff on the API, that's when authorization tokens come in. Each API key has a list of authorization tokens which contain a regex in the URL and available HTTP methods. For example:

```
{
        "token":
        [
            {
                "path": "/api/.*",
                "method":
                [
                    "GET",
                    "POST",
                    "PUT"
                ]
            }
        ],
        "key":
↪"BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8
↪"
}
```

Here we have an API key containing the key and token. The key is just 100 pseudo-random numbers and letters, this key should travel as an HTTP header named 'key' in the request. The other part is the token, it consists of a list where each object in that list consist of a dictionary with a 'path' and 'method'. The 'path' is a regex applied over the requested URL, and 'method' is a list of allowed HTTP methods over that regex match. Our request should match some object on this list, the following example shows a positive authentication.

```
GET /api/hq/rules HTTP/1.1
key:␣
↪BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8
Content-Type: application/json
```

This example shows a denied authorization.

```
DELETE /api/hq/rules HTTP/1.1
key:␣
↪BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8s
Content-Type: application/json
```

With this scheme one can assign API keys to both users and scripts, therefore a user can easily use this API (ie. Postman) and also a Python script (ie. with Requests.

Add a user

To add a new user to the API use the configured user and password for admin access (located here) as HTTP authentication. Make a GET to /keymgmt.

```
GET /keymgmt HTTP/1.1
Authorization: Basic YWRtaW46c2VjcmV0
Content-Type: application/json
```

If you never added a user to the API this request should return an empty JSON. If not, it will return a JSON dictionary of user numbers and their respective key and tokens.

```
{
   "1" :
   {
           "comment" : "Audit"
           "token":
           [
              {
                 "path": "/api/.*",
                 "method": [
                    "GET",
                    "POST",
                    "PUT",
                    "PATCH",
                    "DELETE"
                 ]
              }
           ],
           "key":
↪"BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8
↪"
       },
       "2" :
       {
          "comment": "NOC",
          "token":
          [
              {
                 "path": "/api/hq/.*",
                 "method": [
                    "GET"
                 ]
              },
```

```
            {
               "path": "/api/branch1/.*",
               "method": [
                  "GET"
               ]
            }
         ],
         "key":
↪"xTYRt9tKODjh42smjmoHno3j10OD3LGM3dZgHcen1S5NhICCRzdlrj6VJJwBpBTVgXmfpI3S63bo8aBGZT1CGR91rroBvTv8ce
↪"
   }
}
```

To add a user you need to generate a new pseudo-random API key.

```
POST /keymgmt/generate HTTP/1.1
Authorization: Basic YWRtaW46c2VjcmV0
Content-Type: application/json
{"comment" : "Some User"}
```

```
201 CREATED
```

```
{
     "3": {
            "comment": "Some User",
            "token": [],
            "key":
↪"xWCALV3fPLqnUZ8avZaCeDGyXhTwrTSEMcf7iH7o1j6XG2gGJF75kAXk0l8b2GMsrvHELrXS1T8S4tjfN2SQB2RVH13B0gzGa
↪"
     }
}
```

And now assign new tokens to that user.

```
POST /keymgmt/3 HTTP/1.1
Authorization: Basic YWRtaW46c2VjcmV0
Content-Type: application/json
{
"path": "\/api\/hq\/rules\/.*",
"method": [
  "GET",
  "POST"
]
}
```

```
201 CREATED
```

```
{
     "path": "/api/hq/rules/.*",
     "method": [
            "GET",
            "POST"
            ]
}
```

Take note of the backslash. Check that it was successfull with GET.

```
GET /keymgmt/3 HTTP/1.1
Authorization: Basic YWRtaW46c2VjcmV0
Content-Type: application/json
```

```
200 OK
```

```
{
     "3": {
               "comment": "Some User",
               "token": [
                        {
                        "path": "/api/hq/rules/.*",
                        "method": [
                          "GET",
                          "POST"
                        ]
                        }],
               "key":
↪"xWCALV3fPLqnUZ8avZaCeDGyXhTwrTSEMcf7iH7o1j6XG2gGJF75kAXk0l8b2GMsrvHELrXS1T8S4tjfN2SQB2RVH13B0gzGa(
↪"
               }
}
```

You can't delete specific authorizataion tokens, you would have to delete the entire API key and start over. For that one can use the DELETE method.

```
DELETE /keymgmt/3 HTTP/1.1
Authorization: Basic YWRtaW46c2VjcmV0
Content-Type: application/json
```

```
200 OK
```

CHAPTER 5

Firewall management

This is the second part of the admin configuration, this part should be accessed through HTTP authenteication with the user and password specified in assimilator.conf file. Here the admin configures all Firewall credentials, with this information Assimilator will then access each Firewall and retrieve the information requested through API calls. Each Firewall brand has their our way to be accessed, in general it's an SSH connection but some of them use an API (PaloAlto or AWS).

Add a Firewall

To add a Firewall we make an admin POST request to /firewalls/<firewall key>, in the request's body we should send the JSON object with the Firewall's credentials.

```
POST /firewalls/argentina HTTP/1.1
Content-Type: application/json
Authorization: Basic YWRtaW46c2VjcmV0
{
        "brand" : <firewall brand>,
        "description" : <Some description about this device>,
        #JSON object keys for the Firewall brand
        ...
}
```

To remove a Firewall from Assimilator we make a DELETE request.

```
DELETE /firewalls/argentina HTTP/1.1
Content-Type: application/json
Authorization: Basic YWRtaW46c2VjcmV0
```

To retrieve the Firewall configuration we make a GET request.

```
GET /firewalls/argentina HTTP/1.1
Content-Type: application/json
Authorization: Basic YWRtaW46c2VjcmV0
```

Each Firewall brand is configured differently, this is because each Firewall has their way to be accessed. For each Firewall there is a unique JSON object format. Below is the detailed configuration for each device.

Palo Alto

PaloAlto firewalls have an XML API that only has the GET method. Through this Assimilator translates it to a friendlier API.

```
GET /firewalls/argentina HTTP/1.1
Content-Type: application/json
Authorization: Basic YWRtaW46c2VjcmV0
```

```
200 OK
```

The key is the Firewall name through the api, in this example the key is 'argentina'. Inside this JSON object we have the following keys:

```
"brand" : The Firewall's brand, this will indicate which translator script should be␣
↪invoked when connecting to this firewall.
"primary" : The Firewall's primary IP address, in PaloAlto this should be the␣
↪Management IP address.
"secondary" : The Firewall's secondary IP address, in PaloAlto this should be the␣
↪Management IP address.
"key" : XML API key to be used by Assimilator when connecting to this PaloAlto␣
↪Firewall.
"description" : Some description about this device.
```

Juniper

Junos SRX and SSG have a similar configuration, both are XML based and are accessed through SSH.

```
GET /firewalls/datacenter HTTP/1.1
Content-Type: application/json
Authorization: Basic YWRtaW46c2VjcmV0
```

```
200 OK
```

The key is the Firewall name through the api, in this example the key is 'datacenter'. Juniper allows users to login either with a password or a certificate, the latter one is encouraged. Inside this JSON object we have the following keys:

```
"brand" : The Firewall's brand, this will indicate which translator script should be␣
↪invoked when connecting to this firewall.
"primary" : The Firewall's primary IP address, in Juniper this should be the trust IP␣
↪address.
"secondary" : The Firewall's secondary IP address, in Juniper this should the trust␣
↪IP address.
"user" : The username that Assimilator should use while logging in, it usually is
↪'assimilator'.
"privatekey" : Location of the certificate file to be used for SSH authentication, if␣
↪not specified then user/password will be used.
"privatekeypass" : The password to decrypt the private key from the certificate, if␣
↪not specified then user/password will be used.
```

```
"pass" : The password to be used for SSH login, this is used if privatekey and␣
↪privatekeypass is not specified.
"port" : The SSH port on the Firewall, usually 22.
"description" : Some description about this device.
```

CHAPTER 6

API

The juice of Assimilator relies on the /api. From here one can access all Firewall configuration, check rules, routes and network objects. Also the user can test an access to see if the Firewall grants the access. Assimilator has default resource URL for all firewalls (like rules, objects and routes) and private resource URL destined for each Firewall brand. This is to grasp the full functionality of Firewalls.

Config

/api/<firewall>/config

Gets the full configuration of the Firewall, in it's native format. In many cases this is XML.

Example

```
GET /api/argentina/config
key:␣
↪BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8s
Content-Type: application/json
```

```
200 OK
```

Rules

/api/<firewall>/rules

Get all rules in the selected Firewall. This can be filtered with URL arguments.

Example (PaloAlto)

```
GET /api/argentina/rules
key:␣
↪BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8s
Content-Type: application/json
```

```
200 OK
```

Example with arguments (PaloAlto)

```
GET /api/argentina/rules?from=dmz&to=untrust
key:␣
↪BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8s
Content-Type: application/json
```

```
200 OK
```

To add a rule one simply change the method to POST and sends one of these JSON objects in the body of the request.

```
POST /api/brasil/rules
key:␣
↪BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8s
Content-Type: application/json
{
        "log-end": true,
        "qos": {
                "marking": null,
                "type": null
        },
        "negate-source": false,
        "disabled": true,
        "rule-type": "universal",
        "tag": [],
        "log-start": false,
        "hip-profiles": [],
        "negate-destination": false,
        "description": null,
        "category": [
                "any"
        ],
        "from": [
                "dmz"
        ],
        "service": [
                "any"
        ],
        "source": [
                "any"
        ],
        "destination": [
                "10.10.50.2",
        ],
        "application": [
                "web-browsing",
                "ssl"
        ],
        "profile-setting": null,
        "log-setting": null,
```

```
        "to": [
                "untrust"
        ],
        "schedule": null,
        "source-user": [
                "any"
        ],
        "icmp-unreachable": false,
        "name": "Internet access",
        "disable-server-response-inspection": false,
        "action": "allow"
}
```

To delete a rule, use DELETE.

```
POST /api/brasil/rules
key:␣
↪BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8s
Content-Type: application/json
{
        "name" : "Some Rule Name"
}
```

To replace rules values use PUT, here we **replace** the values of 'source' with new values.

```
POST /api/brasil/rules
key:␣
↪BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8s
Content-Type: application/json
{
        "name" : "Some Rule Name",
        "source" :
        {
                "192.168.1.50",
                "192.168.1.40"
        }
}
```

To append new objects to a rule use PATCH, here we **add** objects to destination.

```
POST /api/brasil/rules
key:␣
↪BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8s
Content-Type: application/json
{
        "name" : "Some Rule Name",
        "destination" :
        {
                "100.200.100.10"
        }
}
```

Match

/api/<firewall>/rules/match

A very useful resource is match. With it one can test a source, destination and port to check if the Firewall allows that connection. Many Firewalls already have this funcionality, other don't (AWS). What they lack is the ease of use. Assimilator only requires source, destination and port (optionally a protocol), other required input by the Firewalls (such as dmz zones) are resolved by Assimilator either through route tables or configuration. If the access is granted then it returns the rule that allows it.

```
GET /api/uruguay/rules/match?source=192.168.4.5&destination=100.150.100.150&port=443
key:␣
↪BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8s
Content-Type: application/json
```

```
200 ok
```

Objects

/api/<firewall>/objects/<address|address-group|service|service-group>

Firewall objects identify hosts and ports in the rules, basically there are four type of objects:

- Address: Hosts identified by an IP, IP range, subnet or FQDN.
- Service: A combination of protocol and source/destination port.
- Address Group: A group of Address objects.
- Service Group: A group of service objects.

With Assimilator one can create/modify/delete objects easily.

```
POST /api/chile/objects/address
key:␣
↪BDP0NyHZMDfz98kcmD3GuBIQGW9EZTgWGPf56dWnkD3LGM3dZPaZICrKVnTnQWh5YdGLh5SJ9ktg7ReR4le94zyxdigdLTHHf8s
Content-Type: application/json
{
        "name" : "Corp_DNS",

}
```

Routes

/api/<firewall>/route

CHAPTER 7

Indices and tables

- genindex
- modindex
- search